ROART: Range-query Optimized Persistent ART

Shaonan Ma\textsuperscript{1}, Kang Chen\textsuperscript{1}, Shimin Chen\textsuperscript{2,3}, Mengxing Liu\textsuperscript{1}, Jianglang Zhu\textsuperscript{1}, Hongbo Kang\textsuperscript{1}, Yongwei Wu\textsuperscript{1}

\textsuperscript{1}Tsinghua University, \textsuperscript{2}SKL of Computer Architecture, ICT, CAS, \textsuperscript{3}University of Chinese Academy of Sciences
Persistent Indexes

- Building blocks for key-value stores and databases
- ~10 years of simulation-based research
  - NVDIMM or DRAM with latencies
Motivations

- Commercial persistent memory was available in 2019.04
  - Intel Optane DC Persistent Memory
- Previous works almost only focused on performance

Performance is enough?

functionality  performance  correctness
Functionality

- Range queries support
  - Tree-based index naturally support range query

- Variable-sized keys support
  - Requirement of most KV stores (Redis, Memcached) and databases (MySQL, PostgreSQL)

This paper focuses on...
Trie suffers more pointer chasing in range queries
Functionality——Variable-sized Keys Support

- NVTree [FAST’15]
- wB+Tree [VLDB’15]
- FPTree [SIGMOD’16]
- LB+Tree [VLDB’20]
- FAST&FAIR [FAST’18]
Shortcomings in B+Trees, a new structure is necessary

Our choice: Range-query Optimized Adaptive Radix Tree
Performance

PM Performance Characteristics [1]
- PM is 2-3x slower than DRAM
- persistent write has poor scalability of bandwidth

Key: number of store/flush/fence instructions
Our choice: ART-oriented optimization

Correctness

- Programming on PM is error-prone [1,2,3].

- Two aspects
  - Memory Safety
    - Persistent memory leaks during allocation and GC
  - Anomaly Resolution in persistent data structures (e.g. dirty read and lost update, more in paper)

Correctness—Memory Safety

**Volatile Allocator**

1. request

2. allocate

**Persistent Allocators**

- Logging-based
  - Constant time recovery
  - Slow allocation/deallocation
- Post-crash GC
  - Fast allocation/deallocation
  - Slow recovery
Correctness—Performance of Allocators

Comparison of different allocators

Existing persistent allocators have their own drawbacks

Our choice: a better persistent allocator

Recovery time of post-crash GC
Design Choices

Functionality
- Range queries
- Variable-sized keys

Performance
- Reduce PM writes
- Reduce flush/fence

Correctness
- Fast allocation
- Fast recovery

ROART
- ART with leaf compaction
- Entry compression
- Selective metadata persistence
- Minimally ordered split
- DCMM with instant restart
Leaf compaction (LC) compacts the pointers of leaf nodes into a leaf array in the radix tree.

If a subtree of the radix tree has less than or equal to $m$ leaf nodes, the subtree is compacted into a leaf array.

**Disadvantage:** leaf compaction causes key comparison in leaf arrays (e.g. Node F, G)

**Solution:** 16-bit fingerprints with pointers to reduce comparison
More on Performance

Leaf Compaction
• Reduce height of tree -> accelerate traversal
• Reduce pointer chasing in range queries -> accelerate range queries
• Reduce leaf node split -> accelerate insertion

Entry Compression
• Compress 1-byte entry and pointer into an 8-byte field -> reduce persistent instructions (e.g. clwb/sfence)

Selective Metadata Persistence
• Make metadata volatile with constant recovery time -> further reduce persistence overhead

Minimally Ordered Split
• Reorder steps of internal node split -> reduce sfence instructions

More details in the paper
Making ROART Correct

Delayed Check Memory Management (DCMM)
An Allocator based on post-crash GC
DCMM

**Allocation**

- Global PM Manager
  - request
  - Thread-local Allocator 1
    - request
  - Thread-local Allocator 2
    - request
  - Thread 1
  - Thread 2

**Recovery**

- Traverse the index
  - Used chunks
    - L0
      - Thread-local Allocator 1
      - Thread-local Allocator 2
      - Thread-local Allocator N
    - L1
    - L2
  - Calculate unused chunks
    - Thread 1
    - Thread 2
    - Thread N
DCMM recovery is parallel with front-end process operations, with constant recovery time
Evaluation Setup

- Dell PowerEdge R740 server
- Four Intel(R) Xeon(R) Gold 5220 processors supporting clwb
- 6×128GB Optane DC PMM per socket
- 32KB L1-cache, 1MB L2-cache, and 25MB L3-cache
- Persistent memory is managed by a DAX file system

Indexes to Compare

- P-ART [SOSP’19]
- PMwCAS-ART (based on PMwCAS [ICDE’18])
- FAST&FAIR [FAST’18]
- Lock-free SkipList [ATC’18]
- BzTree [VLDB’18]

Some modifications

- We modify P-ART, FAST&FAIR and SkipList with DCMM allocator for a fair comparison.
- PMwCAS and BzTree use their own persistent allocators based on PMDK.
- We also implement some missing operations in these indexes.
Overall Performance

Workloads

• Keys are randomly generated with sizes between 4 to 128 bytes.
• Values are fixed as 8 bytes.
• Each test warms up using 30 million KVs.
• Each test runs 20 seconds for different workloads and reports the average throughput.

Microbench with 4 threads

YCSB
Performance Breakdown

Performance improvement brought by each optimization
Range Query Performance

Performance improvement of range queries from Leaf Compaction
Conclusion

• Analyze three practical aspects
  • functionality
  • Performance
  • correctness

• A new persistent index, ROART, taking all these practical aspects into account.
  • Leaf compaction
  • Entry compression
  • Selective metadata persistence
  • Minimally ordered split
  • DCMM with instant restart

• Better performance than the state-of-the-art persistent indexes
Thanks
Q&A
Anomaly Resolution

Steps of insertion in a lock-free linked list
- Create a new node, set its next pointer, and persist the new node
- Update the next pointer of the predecessor to the new node
- Persist the next pointer of the predecessor

States of the predecessor’s next pointer during three steps

<table>
<thead>
<tr>
<th>next pointer</th>
<th>Step (i)</th>
<th>Step (ii)</th>
<th>Step (iii)</th>
</tr>
</thead>
<tbody>
<tr>
<td>volatile state</td>
<td>old</td>
<td>new</td>
<td>new</td>
</tr>
<tr>
<td>persistent state</td>
<td>old</td>
<td>old</td>
<td>new</td>
</tr>
</tbody>
</table>

Dirty Read

Lost Update

Some existing ways can be used to fix anomalies
- Link-and-persist
- PMwCAS
- non-temporal instructions
Disadvantages of Leaf Compaction

string comparison in leaf array

In ART, search will reach the corresponding leaf. But in ROART, search only reaches a leaf array, and string comparison is necessary.

16-bit fingerprints with pointers to reduce comparison

heavy leaf array split

Leaf array split is rare
Advantages and Disadvantages

### Advantages

- Reduce height of tree ➔ accelerate traversal
- Reduce pointer chasing in range queries ➔ accelerate range queries
- Reduce leaf node split ➔ accelerate insertion

### Disadvantages

- String comparison in leaf array

In ART, search will reach the corresponding leaf. But in ROART, search only reaches a leaf array, and string comparison is necessary.

**Solution**: 16-bit fingerprints with pointers to reduce comparison
Persistency Optimization

**Entry Compression**

- **N4**
  - Header
  - Byte array
  - Pointer array
  - 4 (1-byte) entries (byte)
  - 4 (8-byte) pointers
  - 256 (1-byte) slots

- **N16**
  - Header
  - Byte array
  - Pointer array
  - 16 (1-byte) entries (byte)
  - 16 (8-byte) pointers

- **N256**
  - Header
  - Pointer array
  - 256 (8-byte) pointers

- **N48**
  - Header
  - Child index
  - Pointer array
  - 48 (8-byte) pointers

**Art Node Type**

N4, N16, N48 all have **Byte array (1-byte per entry)** and **Pointer array (8-byte per entry)**. Pointers in current architecture only use 6-byte, so compression can be adopted.

**Selective Metadata Persistence**

- **Metadata (count, lock, offset, etc.)**
  - **Persistent**
  - **Volatile**

**Persistence overhead and recovery time** is a trade-off.

Selective Metadata Persistence (SMP) minimizes the two overhead at the same time.

**Global generation number** is used to check whether metadata of the node needs to be restored when accessing a node. Such a method has no effect on performance.